Emulating Fixed Function Lighting and Beyond

Notes for a Msc Course in Computer Graphics

University of Minho
António Ramires

2013-10-23

Introduction

Lighting is essential in computer graphics. Scenes without lighting seem too flat, making it hard to perceive the three dimensional shape of objects. In here we are going to introduce lighting, replicating the fixed function lighting that is still available, although deprecated, in OpenGL, and go a step further to phong lighting

Lighting is deeply related to colour. When an object is lit we observe colour, otherwise, if no light reaches an object looks completely black. Colour in CG is composed of several terms, namely:

- **diffuse**: light reflected by an object in every direction. This is what we commonly call the colour of an object.
- **ambient**: used to simulate bounced lighting. It fills the areas where direct light can't be found, thereby preventing those areas from becoming too dark.
- **specular**: this is light that gets reflected more strongly in a particular direction, commonly in the reflection of the light direction vector around the surface's normal.
- **emissive**: the object itself emits light

The next figure shows the effect the first three color component when an object is lit. To define an object's material we define values for each of the above components.

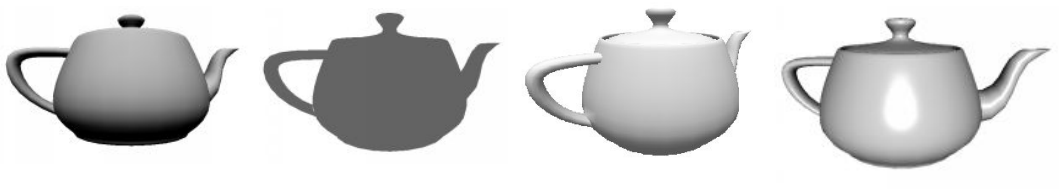


Figure: From left to right: diffuse; ambient; diffuse + ambient; diffuse+ambient+specular

Lights come in many packages as well. The most common light types, and easier to implement, are: **directional**, **point**, and **spot** lights.

In a **directional light**, we assume that all light rays are parallel, as if the light was placed infinitely far away, and distance implied no attenuation. For instance, for all practical purposes, for an observer in planet earth, the light that arrives from the sun is directional. This implies that the light direction is constant for all vertices and fragments, which makes this the easiest type of light to implement.

Point lights spread their rays in all directions, just like an ordinary lamp, or even the sun if we were to model the solar system.

Spotlights are point lights that only emit light in a particular set of directions. A common approach is to consider that the light volume is a cone, with its apex on the light's position. Hence, an object will only be lit if its inside the cone.

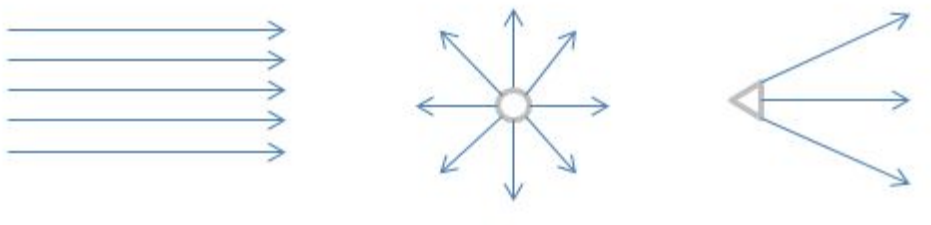


Figure: Lights. From left to right: directional, point, and spotlights.

In the next sections we're going to see how to write shaders to simulate all these types of lights.

Transformations and Interpolations

Spaces and Matrices

Before we start with the lighting shaders we're going to present the spaces, transformation matrices and the interpolation procedure.

The three relevant spaces in CG for our purpose are:

- **local** space: the space where the models are created.
- **global** or **world** space: this is where we assemble our 3D scene
- **camera** space: the space where the camera is at the origin, looking down on the negative Z axis.
-

To transform between these three spaces we have two 4x4 matrices:

model matrix: to transform from local to world space;

view matrix: to transform from world space to camera space.

For our convenience we are also going to consider a matrix called `viewModel` which is the composition of the previous two matrices.

$$viewModel = view \times model$$

Both model and view matrices are commonly constructed based on translation, rotations, and scales. These geometric transformations represent affine transformations, i.e. a transformation that preserves straight lines and ratios - if three points are in a straight line, they will remain in a straight line after being transformed and the transformed midpoint of the line will still be the midpoint of the transformed line. Furthermore, while it does not necessarily preserve angles or lengths, two parallel lines will remain parallel after being transformed.

The matrix form for a general affine transformation, using homogeneous coordinates is a matrix whose last row is $[0\ 0\ 0\ 1]$.

In here, we are only going to consider affine transformations when considering transformations between these three spaces.

Transformation between spaces

To transform a point, a four element tuple $(x,y,z,1)$, between spaces we use the above matrices. For instance, to transform a point P from local to camera space we write:

$$P' = view \times model \times P = viewModel \times P$$

Since we're only considering affine transformations we know that the transformed point will have the general form $[x',\ y',\ z',\ 1]$.

Vectors, four element tuples of the general form $[x,\ y,\ z,\ 0]$, are transformed using the same procedure, and the transformed vector will have the general form $[x',\ y',\ z',\ 0]$. In particular, as the fourth element of a vector is zero, we get an equivalent result using only the first three components of the vector and the top 3x3 submatrix.

The above applies to all vectors that can be expressed as a difference between two points. Considering such a vector, then transforming the vector is equivalent to transforming each of the points, and computing the transformed point difference.

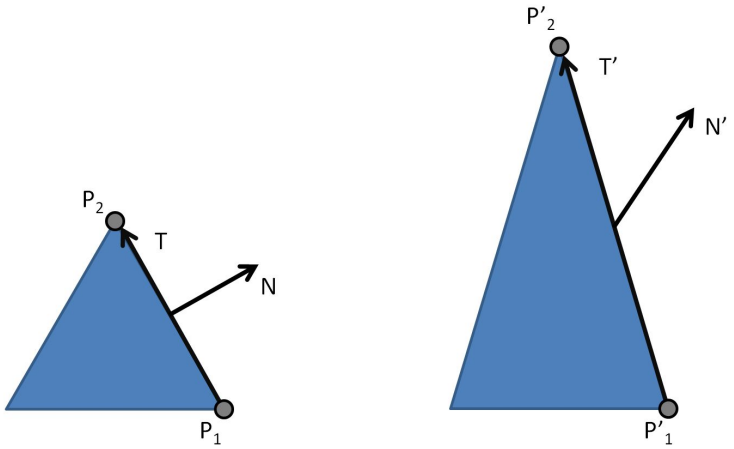
$$v = P_2 - P_1$$
$$v' = M \times v = M \times (P_2 - P_1) = M \times P_2 - M \times P_1 = P_2' - P_1'$$

Normal vectors are a particular case, as they can not be expressed in this way. A normal vector is a vector that has constant magnitude, 1, and a direction which is defined not as the

difference between two points, but as a direction which is perpendicular to a surface.

Hence, to transform the normal vector we must use a matrix that transforms normals from local space to camera space preserving the property of being perpendicular to the transformed surface. So how do we compute such a matrix?

Consider the following vectors in the image below.



On the left is the original triangle and a normal vector to the edge. On the right we have the transformed figure where all points and vectors have been transformed with a scale (1,2,1).

Vector T is tangent to the triangle edge, and can be defined as

$$T = P_2 - P_1$$

As shown above, the transformation of T is equivalent to the difference of the transformed points, i.e.

$$T' = P'_2 - P'_1$$

Hence, the transformed vector, T', remains tangent to the edge. On the other hand, the transformed normal vector is no longer perpendicular to the edge.

As mentioned before, when transforming vectors we can consider only the first three components of the vector, and the top 3x3 submatrix. We need another matrix to transform vector N. A matrix that guarantees that after being transformed N' remains perpendicular to T. Let's call this 3x3 matrix G and let's call the 3x3 matrix that transforms T as M.

We know that after transforming both vectors they must remain perpendicular, hence their dot product must be zero. Hence

$$(MT)\cdot(GN)=0$$

Rewriting the dot product as a matrix product we have

$$(MT)^T\times(GN) = T^T\times M^T\times G\times N$$

If

$$M^T\times G = I$$

Then

$$T^T\times M^T\times G\times N = T^T\times N = T\cdot N = 0$$

Which we know it is true. So we know the relation between G and M is:

$$M^T\times G = I$$

Therefore, multiplying by the inverse of the transpose of M on both sides we get

$$(M^T)^{-1}\times M^T\times G = (M^T)^{-1}$$
$$G = (M^T)^{-1}$$

Therefore, the normal matrix, matrix G, must the the inverse of the transpose of M, i.e. the inverse of the transpose of the top 3x3 submatrix of the 4x4 matrix used to transform points.

A matrix is said to be orthogonal if

$$A\times A^T = I$$

or

$$A^T = A^{-1}$$

If M is orthogonal then

$$G = M$$

which can avoid us any computational effort to get matrix G.

An orthogonal matrix has all rows (columns) as unit length, and these are mutually perpendicular. Starting from an orthogonal matrix, such as the identity matrix, the geometric transformations for rotations and translations, preserve these properties. The same does not apply to scales, which change the magnitude of the row (column) vectors of the matrix.

Therefore, if we use only rotations and translations we guarantee that our matrices are always orthogonal, and we can transform all vectors, including normals, with the same matrix.

To ensure that the shaders that will be presented in this chapter can handle scales, we are going to use the normal matrix (3x3) to transform normal vectors (1x3), and regular matrices (4x4) to transform all other vectors (4x1) and points.

Interpolation

As described in the previous chapter, normal vectors require some caution regarding interpolation. All other vectors and points are interpolated correctly. Due to the particular features of normal vectors, these are the only ones which require the unit length property to be guaranteed before and after interpolation. This can be guaranteed through normalization of the normal vector.

Directional Lights

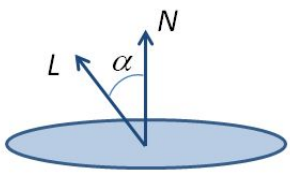
We are going to consider two situations regarding the distribution of the computation amongst the shaders (vertex and fragment): Initially we are going to write as much of the computation as possible in the vertex shader, and then we'll reverse the situation. This has major implications in the rendering quality.

When rendering an object we need uniform variables to define the light and the material. A directional light is defined solely by its direction, hence a `vec4` is sufficient. We could also define a colour for the light, but as this does not bring any added value to the examples presented next, we're going to assume the light is white, as in (1.0, 1.0, 1.0, 1.0).

Regarding the material, we are going to start with the diffuse component, and gradually we'll add the other components.

In this first version, we are going to define the following uniforms: the light direction (`vec4`) and the diffuse component of the object's material (`vec4`).

To compute the intensity of the reflected light, assuming a Lambertian diffuse reflection model, we compute the cosine between the vector pointing towards the light and the normal vector. Hence, we can compute the dot product between these two vectors, assuming that the vectors are normalised and they are defined in the same space. The result of the dot product then gets multiplied by the diffuse component of the object's material, K_d. The lighting equation is:



$$I = K_d \times cos(\alpha) = K_d \times (N \cdot L)$$

Its important to stress that in order to perform operations with multiple vectors, these must be in the same space. There are typically many spaces where we can work on. Local space, world space, and camera space, are the most common to work with vectors. Regarding lighting, we can define the light's direction (and position later on) in any of these spaces. When we define a light in local, or model, space, it works as if the light is attached to the object, like the light bulb in a desk lamp, as in Luxo Jr. from Pixar. Defining a light in world space works as if the light is fixed in the 3D world we are building, regardless of the camera or any object. Working with lights in camera space means that the light is defined relatively to the camera (which in this space is placed at the origin, looking down on the negative Z axis). Whenever the camera moves, the light follows. A miner's helmet lamp, fixed to the helmet, is an example of such light, if we consider the eyes of the miner to be the camera.

For shaders to work with multiple lights, potentially in multiple spaces, it makes life easier for the shader programmer to assume that all the lights are defined in the same space. Camera space is a common option.

Hence, we either transform the lights properties, such as direction and position, in the application and send these values in the common space to the shader, or we'll have to consider where the light has been defined and transform those properties accordingly.

Assuming that we're going to work in Camera space, the normals must be transformed from local space to camera space. As mentioned before we are going to use the normal matrix for this purpose.

In here we are going to assume that the all the light's data fed to the shader is in world space. This implies that the vector representing the light's direction (actually the vector points to the light) must be transformed by the view matrix.

In all shaders, the matrix `pvm` (`mat4`) stands for a matrix that is computed as the multiplication of the projection, view, and model matrices. The normal matrix is referred to as `normal` (`mat3`), and the view matrix as `view` (`mat4`).

In our first solution we're going to compute a color, or reflected intensity, per vertex, using the above equation.

The shader must receive as inputs the position and normals of each vertex, and output the computed color. We also need the above mentioned matrices and the direction towards the light, `lightDir` (`vec4`), and the diffuse color of the material, `diffuse` (`vec4`).

Although it may look excessive in this first example, we're going to use uniform blocks for both the matrices,material, and light properties as these will help to keep the code clean later on.

We are also going to use blocks for intershader communication.

Vertex Shader

```
#version 330

layout (std140) uniform Matrices {
	mat4 pvm;
	mat4 viewModel;
	mat4 view;
	mat3 normal;
} Matrix;

layout (std140) uniform Materials {
	vec4 diffuse;
	vec4 ambient;
	vec4 specular;
	vec4 emissive;
	float shininess;
	int texCount;
} Material;

layout (std140) uniform Lights {
	vec4 dir; // world space
} Light;

in vec4 position; // local space
in vec3 normal;    // local space

out Data {
	vec4 color;
} DataOut;


void main () {
	// transform both vectors to camera space
	// and normalize them
	vec3 n = normalize(Matrix.normal * normal);
	vec3 l = normalize(vec3(Matrix.view * Light.dir));

	// compute the intensity as the dot product
	// the max prevents negative intensity values
	float intensity = max(dot(n, l), 0.0);

	DataOut.color = intensity * Material.diffuse;

	gl_Position = Matrix.pvm * position;
}
```

The fragment shader has a very simple job: receiving the color computed in the vertex shader and outputting it.

Fragment Shader

```
#version 330

in Data {
	vec4 color;
} DataIn;

out vec4 colorOut;

void main() {

	colorOut = vec4(DataIn.color);
}
```

The result using the above shaders is presented in following figure. The lit surfaces look nicely curved, but the surfaces facing away from the light are completely dark.

Figure: Directional light and diffuse material

Let's add the ambient component.

The ambient term is a constant that is added to the previously computed color. It prevents unlit areas from becoming too dark. The new lighting equation is as follows:

$$I = K_d \times cos(\alpha) + K_a$$

The vertex shader needs a new uniform variable, in the `Materials` block, to hold the ambient term. The new block looks is as follows:

```
layout (std140) uniform Materials {
	vec4 diffuse;
	vec4 ambient;
} Material;
```

As for the main function, there is only a small change, the color computation:

```
DataOut.color = intensity * Material.diffuse + Material.ambient;
```

The result is as presented in the next figure, considering a diffuse term of (0.6, 0.8, 0.6), and an ambient term of (0.2, 0.266, 0.2).

Figure: Directional light with diffuse and ambient components

Actually, looking at both the diffuse and ambient terms, we can see that the ambient term could be computed as ⅓ of the diffuse term. Hence, if this is the case, instead of adding a new variable to the computation of the color we could have written:

```
DataOut.color = (intensity + 0.33) * Material.diffuse;
```

In some models this might cause the model to become too bright, as the ambient term is also being added to lit areas. A possible work around, which actually makes more sense, is to consider the ambient component as a minimum lighting intensity:

```
DataOut.color = max(intensity * Material.diffuse, Material.ambient);
```

or, when considering the ambient term as a scaled down diffuse term:

```
DataOut.color = max(intensity * Material.diffuse,
                        0.25 * Material.diffuse);
```

Of course, there is always the possibility of adjusting the color terms themselves, but when considering a large number of models a programmable approach might be preferable.

Adding the specular term

When we consider shiny materials we see that there is a bright spot, sometimes in a colour different from the diffuse colour. For instance an apple may be green, but the bright spot is white. This bright spot varies in size, being more sharp in metallic objects, and more diffuse in plastics. The position and intensity of the bright spot varies with the position of the observer.

The intensity of the reflected specular light is at its maximum in the direction of the reflection of *L* around *N*. The reflection vector can be computed as in the following diagram:

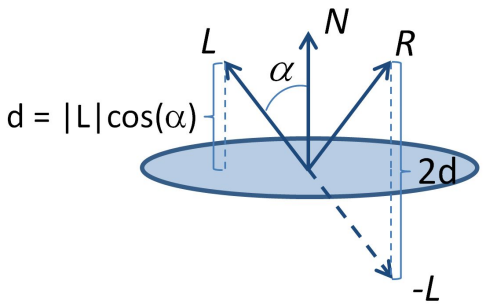


Figure: Computing the reflection vector

We know that $|L|$ is one, and $cos(\alpha)$ is simply the dot product between *L* and *N*. Going from the end of *-L* to the end of *R* one needs to go in the *N*'s direction, with a magnitude twice of $|L|\,cos(\alpha)$. The equation to compute *R*, according to the diagram in the figure above is as follows:

$$R = -L + 2(N.L)N$$

The intensity of the bright spot will have its maximum magnitude when the camera vector is aligned with the reflection vector. As the camera vector moves away from the reflected vector, the intensity of the bright spot will become dimmer.

Phong proposed that the intensity of the bright spot be computed as the cosine of the angle between the reflection vector and the eye vector, as depicted in the next figure.

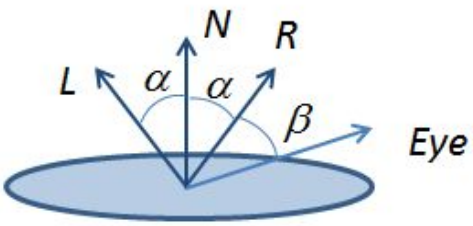


Figure: Specular lighting

Phong's specular term captures this phenomenon in the following equation:

$$I_s = K_s \times cos(\beta)$$

Blinn later proposed an alternative to the Phong equation using the half-vector. The half-vector is the vector that its half way between the light vector and the eye vector. The vectors are depicted in the following figure.

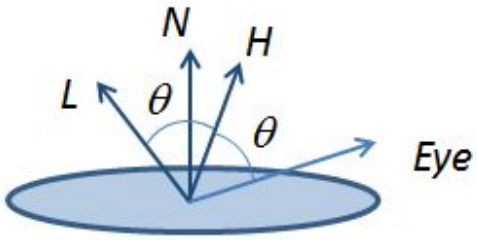


Figure: Half-Vector

The half-vector, *H*, is computed as

$$H = L + Eye$$

The cosine between the vector *H* and the normal vector, *N*, provides an alternative Phongs original idea. Note that to use the dot product to compute the cosine the vector H must have been normalized previously. The equation with the half-vector is called the Blinn-Phong equation.

$$I_s = K_s \times cos(N, H)$$

However applying this equation directly can cause the specular light to flood the scene, washing away all the other colors, as shown in the next figure:.

Figure: Applying the specular term

To solve this we're going to add another term, called shininess, to control the specular term. The equation with this new term, *s*, is as follows:

$$I_s = K_s \times cos(N, H)^s$$

The result of adding the shininess term can be observed in the following figure. For *s* = 1, we have the cosine curve. As s grows bigger, the curve gets steeper, essentially providing non zero values for the specular term in smaller and smaller intervals. On the other hand, if *s* goes below 1, then the curve gets wider, hence the specular term will affect almost all points.

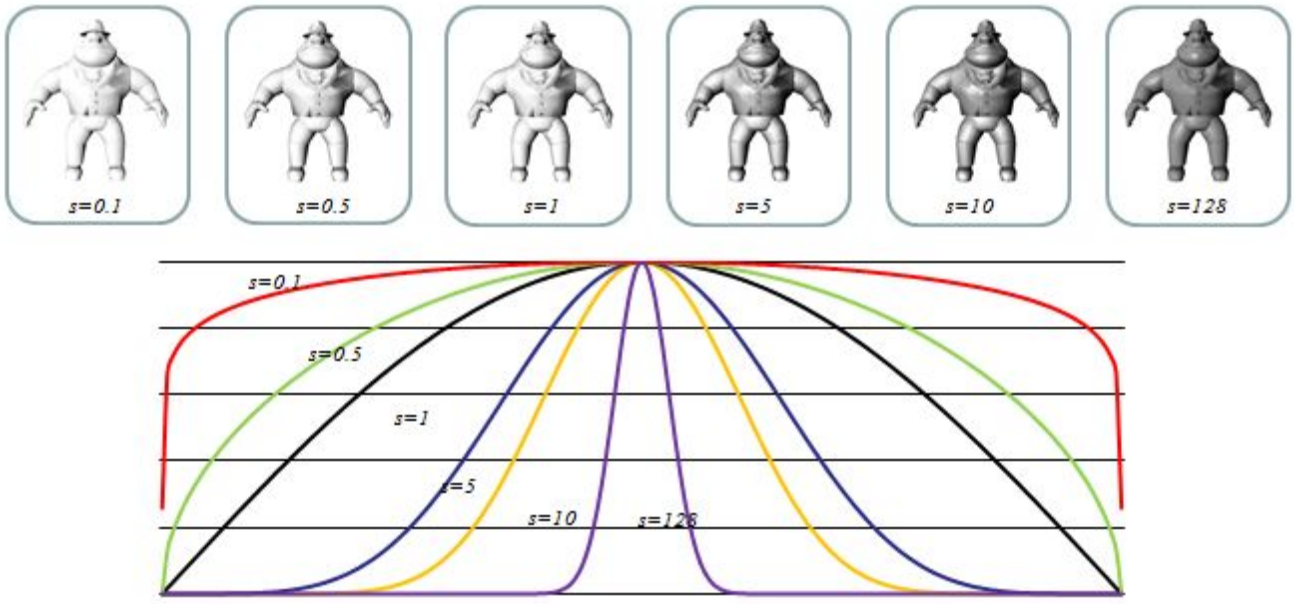


Figure 8. Applying the shininess term

Getting back to our teapot, when s = 100 we get the following result:

Figure: Teapot with specular term and shininess set to 100

Now, let's see how to implement this. We need two more uniforms in our `Materials` block, the specular color, and the shininess term.

Inside the main function we have to compute the eye vector, and the half-vector. To compute the eye vector we have to transform the vertex's position with the View Model matrix (`viewModel` in the shader) to define it in Camera space.

Then, we compute the dot product between the normalized half-vector and the normal to determine the specular intensity. We use the power function with the specular intensity and the shininess term, and finally multiply it by the specular color. The specular color is then added to the final color.

The following vertex shader implements these concepts:

```
#version 330

layout (std140) uniform Matrices {
	mat4 pvm;
	mat4 viewModel;
	mat3 view;
```

```
	mat3 normal;
} Matrix;

layout (std140) uniform Materials {
	vec4 diffuse;
	vec4 ambient;
	vec4 specular;
	float shininess;
} Material;

layout (std140) uniform Lights {
	vec4 dir; // world space
} Light;

in vec4 position; // local space
in vec3 normal;    // local space

out Data {
	vec4 color;
} DataOut;


void main () {

	vec4 spec = vec4(0.0);
	vec3 n = normalize(Matrix.normal * normal);
	vec3 l = normalize(vec3(Matrix.view * Light.dir));
	float intensity = max(dot(n, l), 0.0);

	if (intensity > 0.0) {
		vec3 pos = vec3(Matrix.viewModel * position);
		vec3 eye = normalize(-pos);
		vec3 h = normalize(l + eye);

		float intSpec = max(dot(h,n), 0.0);
		spec = Material.specular * pow(intSpec, Material.shininess);
	}

	DataOut.color = max(intensity *  Material.diffuse + spec,
	                    Material.ambient);

	gl_Position = Matrix.pvm * position;
}
```

This implementation provides an implementation of the Blinn-Phong equation per vertex, with fragments getting interpolated colors. This corresponds to the **Gouraud lighting model**.

This model has some drawbacks, since interpolating colors is not the correct way of

obtaining the colors per fragment as seen in the toon shader example.

Lighting per Pixel

Next, we're going to modify our example to use **Phong lighting model**, i.e. we are going to compute the color per fragment. This implies moving most of the operations from the vertex shader to the fragment shader. The vertex shader limits itself to prepare the data for the computations that will take place in the fragment shader.

The fragment shader will need the following data per fragment:

- normal
- light direction: vector towards the light
- eye vector: vector from the point to the eye

The vertex shader must compute these vectors per vertex, so that they get interpolated and passed on to the fragment shader.

The new vertex shader is:

```
#version 330

layout (std140) uniform Matrices {
	mat4 pvm;
	mat4 viewModel;
	mat4 view;
	mat3 normal;
} Matrix;

layout (std140) uniform Lights {
	vec4 dir;
} Light;

in vec4 position;
in vec3 normal;

out Data { // all computed in camera space
	vec3 normal;
	vec3 eye;
	vec3 lightDir;
} DataOut;

void main () {

	DataOut.normal = normalize(Matrix.normal * normal);
	DataOut.lightDir =  vec3(Matrix.view * Light.dir);
	DataOut.eye = -vec3(Matrix.viewModel * position);
```

```
	gl_Position = Matrix.pvm * position;
}
```

The fragment shader receives the interpolated vectors and performs the remaining computations, as follows:

```
#version 420

layout (std140) uniform Materials {
	vec4 diffuse;
	vec4 ambient;
	vec4 specular;
	float shininess;
} Material;

in Data {
	vec3 normal;
	vec3 eye;
	vec3 lightDir;
} DataIn;

out vec4 colorOut;

void main() {

	vec4 spec = vec4(0.0);

	vec3 n = normalize(DataIn.normal);
	vec3 l = normalize(DataIn.lightDir);
	vec3 e = normalize(DataIn.eye);

	float intensity = max(dot(n,l), 0.0);

	if (intensity > 0.0) {
		vec3 h = normalize(l + e);
		float intSpec = max(dot(h,n), 0.0);
		spec = Material.specular * pow(intSpec,Material.shininess);
	}
	colorOut = max(intensity * Material.diffuse + spec,
Material.ambient);
}
```

The following figure shows the difference between the Gouraud and Phong lighting models.

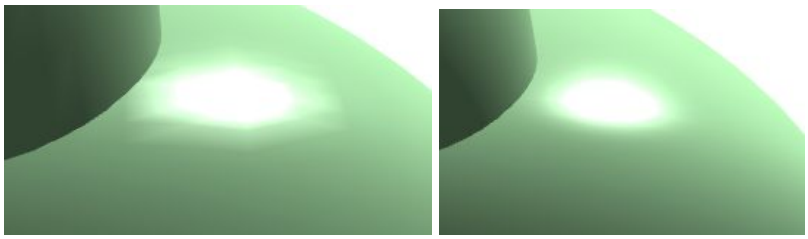


Figure: Gouraud (left) and Phong (right)

Point Lights

When considering point lights, the main difference is that now we have a position for the light instead of a direction. This implies that the light direction is not constant across all fragments/vertices as it was for direction lights. Besides that, everything remains just like for directional lights.

The vertex shader now receives a light position instead of a light direction, and it must compute the light's direction for each vertex. It is assumed that the light's position is in world space. Once we move both the light position and vertex position to the same space, the direction computation is straightforward: *light position - vertex position*.

If both these positions were in camera space, the resulting vector would also be in camera space which is what we want since most of our computations are in camera space. To achieve this we must apply the View Model matrix (`viewModel` in the shader) to the vertex position, and the View Matrix (`view` in the shader) to the light's position..

Vertex Shader

```
#version 330

layout (std140) uniform Matrices {
	mat4 pvm;
	mat4 viewModel;
	mat4 view;
	mat3 normal;
} Matrix;

layout (std140) uniform Lights {
	vec4 pos;
} Light;

in vec4 position;
in vec3 normal;

out Data {
	vec3 normal;
```

```
	vec3 eye;
	vec3 lightDir;
} DataOut;

void main () {

	vec4 pos = Matrix.viewModel * position;
	vec4 lpos = Matrix.view * Light.pos;

	DataOut.normal = normalize(Matrix.normal * normal);
	DataOut.lightDir = vec3(lpos-pos);
	DataOut.eye = vec3(-pos);

	gl_Position = Matrix.pvm * position;
}
```

Since the light direction is computed in the vertex shader, and it is then interpolated, the fragment shader is identical to the directional light case.

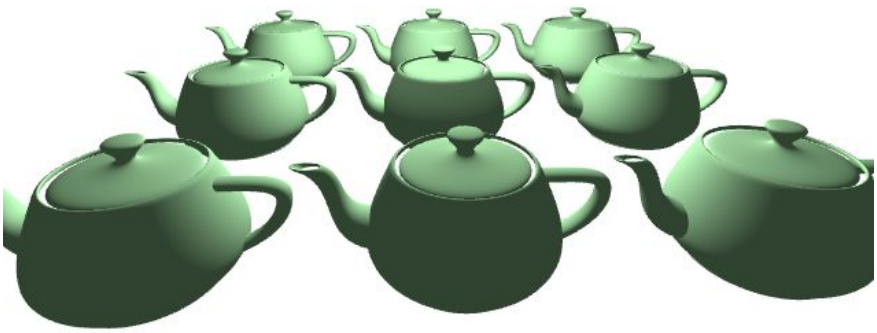


Figure: a point light placed above the center teapot.

Spotlights

Spotlights are restricted point lights, i.e. the light rays are only emitted in a restricted set of directions. Commonly we use a cone to define this restriction, but other shapes are possible.

Considering the cone as an option to restrict the light rays, we need the following data to define a spot light:

- position: this is the cone's apex
- direction: the vector that defines the direction of the axis of the cone
- angle: the aperture of the cone. In here we are going to assume that the angle is

measured from the direction vector to the border of the cone.

The vertex shader is the same as for the point light. It is up to the fragment shader to determine if a fragment is inside the cone, i.e. the dot product between the light's direction and the spot's direction is less than some cutoff value, and lit it accordingly.

Fragment Shader

```
#version 150

uniform vec3 diffuse;
uniform vec3 specular;
uniform vec3 spotDir;
uniform float cutOffAngle;

in vec3 normalV;
in vec3 eyeV;
in vec3 lightDirV;

out vec3 colorOut;

void main() {

	vec3 sd = normalize(-spotDir);
	vec3 ld = normalize(lightDirV);
	vec3 n = normalize(normalV);
	vec3 e = normalize(eyeV);

	if (acos(dot(sd, ld)) < cutOffAngle) {

		float intensity = max(dot(n,ld), 0.0);
		vec3 h = normalize(ld + e);
		float intSpec = max(dot(h,n), 0.0);
		colorOut = (intensity + 0.33) * diffuse  +
				specular * pow(intSpec,100);
	}
	else
		colorOut = 0.33 * diffuse;
}
```

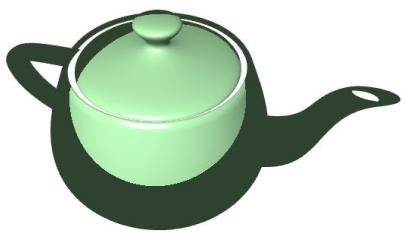


Figure: Spotlight in teapot

Two notes regarding the implementation. First, we used -spotDir because lightDir is also reversed, i.e. it is not the vector with a direction from the light source to the point being lit, but the reverse. Secondly, we could have save ourselves the cost of computing the arc of the cosine, if we assumed that the spotlight's aperture is specified as a cosine, instead of an angle (in radians). The user can still specified it as an angle, but it would be up to the application to compute the cosine before feeding the value to the shader.